

Push Job Specification for V1 Product

Mark Anderson <mark@opscode.com>

Christopher Brown <cb@opscode.com>

Kevin Smith <kevin@opscode.com>

June 19, 2012

Contents

1 Overview	2
1.1 A simple use case	2
1.2 Internals	3
2 Assumptions	3
2.1 Connectivity	3
2.2 Data format & Storage	4
2.3 Scalability & Security	4
3 Architecture	4
3.1 Communications	4
3.2 Configuration/Discovery Process	6
3.3 General ZeroMQ Messaging	7
4 Heartbeat	9
4.1 Server Heartbeat	9
4.2 Server Heartbeat Message	9
4.3 Node Heartbeat	10
4.4 Node Heartbeat Message	10
4.5 Node monitoring of server heartbeat	10
4.6 Server monitoring of client heartbeat	11
4.7 Basic heartbeat state machine	11
4.8 Simple protocol for detection of crashed node	11



OPSCODE

5 Command Execution	12
5.1 Vocabulary	12
5.2 Overall communications structure	12
5.3 Job lifecycle	13
5.4 Command Vocabulary	16
5.5 Client and server messages	17
5.6 Client command state machine	19

1 Overview

Opscode Chef is currently missing a key feature that comes up in discussion with nearly every potential customer. That feature is immediate, ad-hoc task execution against one or more managed devices (nodes). In Chef, we will model these tasks as *run lists* and will model their execution by distributed, coordinated *chef-client* runs. For brevity's sake this feature will be referred to as *push jobs*. The concept of *push jobs* is quite simple. A user is able to select some subset of nodes managed by their Private Chef organization, specify an action or command for those nodes to execute, and track the status of each node as it executes the request.

1.1 A simple use case

Imagine that we are administering a large collection of machines running various web stacks. We need to patch some, but not all of our systems. You **could** try:

```
% knife ssh "role:my_role" chef-client
```

Being able to use search to specify the set of nodes is nice, but it gets unwieldy to track things if you have more than a few systems. It gets especially painful if some of the commands fail, or some systems are unreachable.

We want to define a set of nodes, issue commands to them, and take actions on them based on the results of prior commands.

```
% knife push job start "role:my_role" chef-client
Starting job id 235
```

Then the status could be tracked, in detail:

```
% knife push job status 235
Node name  Status    Last updated
foo        Failed    2012-05-04 00:00
bar        Done      2012-05-04 00:01
```



Or in the aggregate:

```
% knife push job status 235 --summary
1  node acknowledged, not started
1  node not available, busy
1  node not available, down
14 nodes started
4  nodes completed with errors
20 nodes completed successfully
```

Or individually:

```
% knife push job status 235 node foo
node foo Failed running job id 235 at 2012-05-04 00:00
```

New jobs can be started from the results of prior jobs.

```
% knife push job start --job_id=235 --job_status=failed chef-client
Starting job id 236
```

1.2 Internals

A bit of complexity and bookkeeping lurks underneath push job's simplicity. Managed nodes will need to reliably monitor the job coordinators to cope with outages. Job coordinators also need to monitor the status of each managed node so commands are only sent to available, responding nodes and to track job progress.

The push job server and the managed nodes ensure mutual liveness via a bidirectional heartbeating system. The push job server sends heartbeat messages to all the managed nodes, and the managed nodes send heartbeat messages to the server.

The remainder of this document will attempt to describe the elements for the V1 release in enough detail to allow a reasonable and scalable implementation.

2 Assumptions

2.1 Connectivity

1. Managed nodes **MUST** be reachable via a TCP-enabled network interface.
2. Managed nodes **MUST** be able to authenticate and connect to the Chef REST API
3. Managed nodes **MUST** be able to connect to the heartbeat and job coordination components inside Chef server.



4. Managed nodes **MAY** be able to **accept** incoming TCP connections. (Deprecate this? The initial protocol doesn't require it.)

2.2 Data format & Storage

1. All messages will be formatted as legal JSON.
2. The database is the canonical store of all application data.

2.3 Scalability & Security

1. Push jobs will be deployed in Private Chef only.
2. Push jobs will not be deployed in Hosted Chef.
3. The design must scale up to 8,000 managed nodes per OPC server.
4. Push jobs will honor the same security guarantees made by the Chef REST API.

3 Architecture

3.1 Communications

Managed nodes and server components will communicate using ZeroMQ messaging. The server will bind its endpoints to well known ports, whose addresses will be part of the client configuration process. The clients will connect to those ports, initiating all communications. The sole exception will be initial configuration, which uses the Chef REST API to discover the addresses and ports for the push job server. This structure simplifies running clients behind NAT routers. It also eases scalability testing; we will want to be able to run many hundreds of nodes on the same machine rather than stand up thousands of individual nodes.

Communication can be separated into two categories: heartbeat and job execution. The heartbeat channel is used by the Chef server to detect when managed nodes are offline and, therefore, unavailable for job execution. Managed nodes also use the heartbeat channel to detect when the server is unavailable.

The job execution channel is used by the Chef server to send job execution requests to managed nodes. Managed nodes use the execution channel to send job-related messages such as acknowledging jobs, sending progress updates, and reporting final results.

3.1.1 Heartbeat Channel

ZeroMQ PUB/SUB (publish and subscribe) sockets are used for the server heartbeat because they automatically and scalably manage the fan-out the server needs to broadcast its heartbeat to all the



nodes. The nodes connect to the server heartbeat socket and subscribe. The nodes do not ACK the server heartbeats, and the server should not expect any.

ZeroMQ PUSH/PULL sockets are used for the client heartbeats to the server because that better suits a model where the clients connect to the server heartbeat input. (TODO: find a better name for that). Earlier versions of this spec had PUB/SUB being used for this process. It would be simpler if the node was able to connect to the server to send heartbeats, rather than requiring the server to bind to the node. The latter would require some sort of handshake on startup to inform the server where to connect. While it is possible to bind the SUB to an address and connect the PUB, this seems to be not recommended (see ZeroMQ guide, 'Getting the Message Out'). However, it seems that multiple PUSH to one PULL is supported, and we can bind the PULL socket to an address without trouble.

In the interest of simplicity, the heartbeat protocol should be as stateless as possible. The server continually broadcasts heartbeat messages to all nodes that are listening. Nodes indicate their availability by sending heartbeat messages to the server. There is no process beyond the ZeroMQ connection to start or stop a connection from the node.

- Future extensions

This might be worth modifying in the future. For example we might want to lower the signature validation load on the server by having a per-session symmetric key for the node heartbeat established at startup. Instead of the somewhat expensive public key signature check we could simply decrypt the packet with the session key and check for sanity. (TODO: think about whether this protocol is actually sane and secure)

The heartbeat protocol can be used to help signal state transitions on both the client and server. Probably the most important of those is detection of restarts and crashes. (TODO: Describe simple protocol for identifying those)

3.1.2 Command Channel

The ZeroMQ ROUTER/DEALER pattern will be used for the command channel. The server will bind a ROUTER socket to a known port, and the clients will connect to this via DEALER sockets. Clients will send signed messages to the server announcing their availability for commands, and signed commands will be routed back to each client.

This pattern requires the client to connect and send a message before any commands can be routed to it, because the routing name used to address a node is taken from the first message recieved.

3.1.3 Future work

The client heartbeat channel and the command channel could be combined, simplifying the pattern. Alternately we may wish to have a separate command channel for each job. This would allow the possibly more chatty command traffic to be moved elsewhere for ease of distribution and workload partitioning.

TODO: ADD PICTURE HERE



3.2 Configuration/Discovery Process

There a substantial amount of data specific to pushy that needs to be distributed to the push job clients. We don't want to have to configure this on every node. Furthermore, much of this data is specific to the chef infrastructure, not the users infrastructure, and may change. We will use a special REST endpoint bootstrap the node up from the minimal static configuration to a usable state. A signed *GET* to this endpoint will retrieve the appropriate configuration information in JSON format.

The configuration and service discovery process will provide the following pieces of data:

- The push job server hostname or address
- The port to subscribe to for server heartbeat
- The port to push node heartbeats to
- The command port to connect to
- The public key of the server
- The lifetime of this configuration information

```
{
  "type": "config",
  "host": "opcl.opscode.com",
  "push_jobs": {
    "heartbeat": {
      "out_addr": "tcp://10.10.1.5:10000",
      "in_addr": "tcp://10.10.1.5:10001",
      "interval": 15,
      "offline_threshold": 3,
      "online_threshold": 2
    },
    "command" : {
      "command_addr": "tcp://10.10.1.5:10001",
    }
  },
  "public_key": "AAAAB3NzaC1kc3MAAACBAIZbwlySffbB
5msSUH8JzLLXo/v03JBCWr13fVTjWYpc
cdbl/xL3IK/Jw8Rm3bGhnpwCAqBtsLvZ
0cqXrc2XuKBYjiKWzigBMC7wC9dUDGwDl
2aZ89B0jn2QPRWZuCAkxm6sKpefu++VPR
RZF+iyZqFwS0wVKtl97T0gwWlZAJYpAAA"
```



```
AFQDIipDNo83e8RRp7Fits0DSy0DCpwAA
AIB01BwXg9WSfU0mwzz/0+5Gb/TMAxfkD
yucbcpJNncpRtr9Jb+9GjeZIBqkBQAqwg
dbEjviRbUAuSawNSCdtMgWD2NXkBKEde",
"lifetime":3600
```

```
}
```

type message type

host sender's host name (Private Chef server)

push_jobs/heartbeat/out_addr URL pointing to the server's heartbeat broadcast service

push_jobs/heartbeat/in_addr URL pointing to the server's node state tracking service

push_jobs/heartbeat/interval Interval, in seconds, between heartbeat messages

push_jobs/heartbeat/offline_threshold How many intervals must be missed before the other end is considered offline

push_jobs/heartbeat/online_threshold How many intervals must be missed before the other end is considered online

push_jobs/command/command_addr URL for command channel (TODO: Presently this is in the heartbeat clause, and should be changed.)

public_key The signing key that the push server will use.

lifetime how long in seconds this configuration is good for. The node should reload the configuration information after this has expired.

We may wish to use the discovery process to handle fail-over to a new server and distribution of nodes among multiple servers. The discovery system would allocate the nodes to various active servers and if a node lost the server heartbeat for a certain length of time (or got a reconfigure command via the command channel) it would reload the configuration and start a connection to the appropriate server. We would also reconfigure after the lifetime of the configuration expires.

3.3 General ZeroMQ Messaging

The basic system message consists of two ZeroMQ frames. The first frame contains a version id and the signed checksum of the second frame. The second frame is a JSON blob. This is used throughout the system for server and client heartbeats, status messages, commands and responses.



Push jobs use JSON because ZeroMQ sends and receives messages as complete frames, without fragmentation. JSON also facilitates easier debugging and maintenance of the system since all messages are textual and human-readable. A binary protocol, such as *Protocol Buffers* or *msgpack*, would be more efficient but would also substantially increase the effort required to debug and support the system. We can discuss those as potential optimizations once the initial system is in place. An simple early optimization would be to send gzip (or lz4, depending on our cpu/size trade space) compressed json instead of the ASCII text.

3.3.1 Header Frame

The header frame is a series of semicolon separated clauses. Each clause is a key-value pair separated by a colon.

Version:1.0;Checksum:fyq6ukIwYcUJ9JI90Ets8Q==

Version a major minor or major minor patch tuple separated expressed as integers separated by '.' characters for the protocol version (1.0 for now) (Should this simply be reduced to an integer?)

Checksum Base64 encoded SHA1 checksum of the second frame, encrypted using the sender's private key. The standard erlang library implements Base64 as specified in RFC4648. This derived from signing protocol that Chef uses for REST requests, except that instead of signing significant headers and the HTTP body, we sign a JSON blob.

3.3.2 JSON

The second frame is a JSON text, as defined by RFC 4627. The exact format varies depending on the particular message being sent. The JSON text itself is not encrypted.

All JSON messages will contain a timestamp field in ISO8601 format. Messages older than a TBD value will be rejected to reduce the window for replay attacks.

NOTE: This provides weaker security than our REST API, where we have the option to use SSL for the exchange. This may be acceptable in a OPC environment where the network is trusted, but that may not be the case for some multitenant users such as resellers. This should receive review before we release V1.

The command strings are of especial concern. If the system uses a limited command vocabulary little interesting information will leak, but if arbitrary commands are allowed there may be sensitive information embedded in the command line.

Erlang message sending sample code

```
Sock = connect_to_server("tcp://some_server:8765"),
Sig = sign_message(JSON),
erlzmq:send(Sock, Sig, [sndmore]),
erlzmq:send(Sock, JSON)
```




3.3.3 Socket configuration

The ZeroMQ messages flowing through the system are time sensitive. For example, if we go too long without receiving a heartbeat, we will be declaring the machine down anyways. There is little value keeping many more packets than the online/offline threshold values. Furthermore, the signing protocol will mandate the rejection of aged packets.

The HWM should be kept small (1 would be a good value); there is no point in storing messages for dead nodes any longer than necessary. ZMQ_SWAP should always be zero. Node failure must be accepted and tolerated. If a node has been marked as down (not reachable), we want to drop any messages destined for that node. This is in keeping with the fail-fast philosophy.

4 Heartbeat

Liveness detection in a distributed system is a notoriously difficult problem. The most common approach is to arrange for two parties to exchange heartbeat messages on a regular interval. Let's call these two parties 'A' and 'B'. Both A and B are considered 'online' while they are able to exchange heartbeat messages. If A fails to receive heartbeats from B for some number of consecutive intervals then A will consider B 'offline' and not route any traffic to B. A will update B's status to 'online' once A starts receiving heartbeats from B again.

The protocol described here is loosely based on the Paranoid Pirate Protocol, but with some embellishments introduced for signing.

The heartbeat server sends out regular heartbeats to managed nodes via ZeroMQ PUB/SUB. Managed nodes send their heartbeats over a separate channel. See the Heartbeat Channel section for a visual representation of the message flows and ZeroMQ sockets.

4.1 Server Heartbeat

The server sends out heartbeat messages at a configurable interval. This simple signed message indicates to the clients that the server is up. The channel is one-way, and no ackno

Future versions of the protocol may use this channel to advertise server failovers, trigger global reconfiguration of clients and other low rate broadcast information.

4.2 Server Heartbeat Message

Server heartbeats are composed of two ZeroMQ frames. The first frame contains the header described above, and the second frame contains the the JSON-formmatted heartbeat payloadL

```
{
  "server": "SERVER",           # The hostname of the server
  "timestamp": "TIMESTAMP",    # RFC 1123 timestamp
}
```



```
"type": "MSGTYPE"                # 'heartbeat' for now
"sequence": SEQUENCE_NUMBER",    # integer sequence number
}
```

4.3 Node Heartbeat

PUSH/PULL sockets are used for the node heartbeat. The node PUSHes heartbeats to the server at the host/port specified in the config data received during discovery. The server will not ACK heartbeats. There isn't any reason we couldn't use the heartbeat to convey extra information. The public key signature-based authentication process for heartbeats already requires a moderate sized payload, so a little extra information seems pretty harmless. This is in contrast to the 1-2 byte sized payload in the paranoid pirate protocol. Possible items to include are:

- The port the command processor is listening on.
- ID and status of the most recently received command.
- Information allowing the detection of crashed nodes

4.4 Node Heartbeat Message

Node heartbeats are comprised of two ZeroMQ frames. The first frame contains the header described above, and the second frame contains the JSON-formatted heartbeat payload:

```
{"node": "node123.foo.com",      # node's host name
 "client": "clientname",        # the client signing the request
 "org": "foo.com",              # orgname of the node
 "timestamp": "Sat, 12 May 2012 20:33:15 GMT", # RFC 1123 timestamp
 "state": "idle"                # The state of a node
}
```

4.5 Node monitoring of server heartbeat

The node will discontinue the heartbeat and note the server as down if the server heartbeat state moves to down, and resume it when the server heartbeat resumes.

A managed node must mark the OPC server as offline when it fails to receive server heartbeats for a consecutive number of intervals equal to `push_jobs/heartbeat/offline_threshold`. A managed node must not attempt to send any data when the server is offline. Any job requests received by the managed node from the offline server which haven't begun execution must be discarded.

After a managed node has marked the server as offline it must receive server heartbeats for a consecutive number of intervals equal to `push_jobs/heartbeat/online_threshold` before marking the



server online. The managed node may resume sending data and accepting job requests from the OPC server at this point.

If the node fails to receive a heartbeat for too long, it will query the configuration interface to receive a possible configuration update. This would allow the system to recover from a failed server.

The node may wish to detect if the HWM is reached on the PUSH socket, since it will block when the HWM is reached. One strategy would be to set the HWM low and have some sort of alarm detect if we are blocked for any length of time. If the HWM is reached, we should declare the server down as if it stopped sending heartbeats.

The node can report a variety of states; see the node state table

4.6 Server monitoring of client heartbeat

The server monitors each client heartbeat and records the state in the database. It uses the same state machine as the client. A node is treated as unavailable for jobs if it's heartbeat status is 'down'.

4.7 Basic heartbeat state machine

TODO: Flesh this out and simplify.

State	Event	Action	New State
Startup			Down
Down	Receive Heartbeat	Set counter to zero	Going up
Going Up	Counter Greater than upward threshold		Up
Going Up	Receive Heartbeat before threshold time	Increment counter	Going up
Going Up	Receive Heartbeat after threshold time	Decrement counter (floor zero)	Going up
Up	Receive Heartbeat before threshold time		Up
Up	Receive Heartbeat after threshold time		
Going Down	Receive		
Down	Recieve Heartbeat		

TODO This state machine needs rework in a serious way.

4.8 Simple protocol for detection of crashed node

It can be helpful to know whether a node has crashed and returned (possibly on different hardware) vs undergone a planned restart. This can be done with a guid (the incarnation id) and simple state file (e.g. /var/pushy/incarnation). On startup, the client will look for the incarnation file, load the incarnation GUID from it, and delete the file. If no file is found, the client will generate a new incarnation GUID. On a clean shutdown the current incarnation GUID is written to the file. The client reports this incarnation



GUID in its heartbeat, and if the incarnation id changes the push job server can recognize this and act accordingly. If a command was in flight the server should record that it ended in an indeterminate state. The server should also maintain an incarnation id to allow the clients to discover a reboot that doesn't trigger the heartbeat mechanism.

5 Command Execution

A OPC server sends requests to execute actions to managed nodes. These requests are called commands. The command server listens on an address specified in the configuration process, and clients connect to that address to receive commands.

TODO: Discuss idempotence of commands; desirable, but not required.

Only one command at a time can be executing on a node. In other words, nodes execute commands serially. This makes it easier to reason about the current state of any node and also avoids any undesired runtime interactions between commands.

5.1 Vocabulary

- Job - A collection of nodes executing the same command. Jobs have persistent state. This state can be queried by users via the knife 'job' command.
- Command - A request to a managed node to perform an action immediately.
- Jobset - the set of nodes in the job (Is there a better name this thing)

5.2 Overall communications structure

The command server will create a ROUTER socket bound to a port, and each client will connect via a DEALER socket. On connection to the ROUTER socket, the client will send a signed message indicating that it is available for commands. The message letting us capture the transient socket name. This will provide a way to map the unique name of the connection to the client in question. Commands will be addressed via that unique name. The client will learn the the address of the ROUTER socket via the discovery system.

The server will need to send a separate message to each client. However the message body (aside from the address packet) will remain the same, and we can reuse the ZeroMQ buffers created and save on the signing cost.

When a command is completed on the node, it sends the command results back to the server using the ROUTER/DEALER connection above.



5.2.1 Open Questions

- What happens to commands sent to permanently dead nodes; does it cause a leak sitting in the buffering. Is dropping and restarting ZMQ a solution?

5.3 Job lifecycle

A job is started by passing a job description, a set of nodes (jobset), and an initiating user id to the pushy server. Each node will flow through a series of states as it executes the job ending in either **OK**, or **FAILED**. These states are final; once a node enters it stays there.

The job description includes a command, and optional values including:

- The quorum required to start the job.
- A timeout for the overall job execution.

Possible extensions include:

- A timeout for execution on a single node.
- A limit on how many nodes can be executing simultaneously. We may want to update a bunch of nodes, but limit how many of them are out of service simultaneously
- A limit on the rate at which nodes may be started. This is to limit load on the server. Some of this can be handled by API throttling, but this allows more fine grained control and can be used to run a job 'in the background'.
- A timeout for how long we wait to achieve quorum
- A timeout for how long to wait for an ACK before marking a node dead (or perhaps for any message from the server)

Throughout the process nodes may drop heartbeat, be marked down and return. The final node state depends on where it was in the execution process.

5.3.1 Specification of the nodeset

Initially the set will be specified explicitly as a list of nodes to the server.

Search, both against the chef index and against state from previous jobs will be implemented via the knife command. The collection of nodes is defined by a standard Chef search criteria. The search is executed against the node index and returns a list of managed nodes satisfying the search criteria. The search criteria should eventually allow up/down state and job execution state as factors. For example it would be quite valuable to run a job against all nodes that match a search and 'succeeded' on job id XXXX.



As an optimization we will eventually want to avoid the round trip transmission of large node lists to the knife command and back to the server by having the server accept search terms directly but that is left for a future version.

Entity groups (named groups of nodes) will greatly ease the specification of node lists, and we may want to implement them as part of the first pushy implementation.

5.3.2 Permission checking

We will check that the user id has permissions to execute a job, and whether they have job execution permissions on each node in the set. The simplest model is to add a group 'push job initiators' to the system, and require the user to be in that group.

In addition we will probably want fine grained administrative control, either at the environment level or per node level. We probably should add a new permission **EXECUTE** to the current CRUDG set to allow fine grained control.

Any node for which permissions are denied will be marked as **FAILED** with 'permission denied' as the reason.

5.3.3 Availability checking

If a node is down at the start of the job, it will be marked as **FAILED** with 'down' as the reason.

The server will send the remaining nodes a **PREPARE** message. Nodes that are available for execution will respond with **ACK** and be marked as **READY**, while those unavailable for execution respond with **NACK**. Nodes that **NACK** will be marked as **FAILED** with 'busy' and the job id in the state. Even though nodes may finish with their current work and become available while the job is still in progress, we are treating **NACK** as a permanent condition for simplicity's sake.

If a node drops heartbeat or otherwise goes down prior to starting it is marked as **FAILED** with reason 'down'.

There are two ways to handle quorum and command execution.

The simplest is to wait a predetermined time for the nodes to **ACK/NACK**. If a quorum is achieved we start, otherwise we fail. If a node fails to **ACK** in time, we mark it timed out, and issue a cancel command to the node.

A more sophisticated version is to track nodes transitioning to **READY**, and when the quorum is reached we start execution. If the quorum is not reached, we never start the job. We still need some sort of timeout for **ACK/NACK**, as some nodes may not transition to ready, holding the job open until it times out or is aborted.

If two users are running jobs against the same set of nodes, some nodes may be affiliated with one job, and some with another. This may cause us to not reach quorum, or run a particular command on one subset and a different command on the rest. The assumption is that it is preferable to make some forward progress rather than inflict a global lock on jobs. If this is an issue it should be avoided by setting high quorum requirements.



5.3.4 Execution

Once quorum is achieved we start sending **START** messages. The naive implementation would just issue messages as quickly as possible to those nodes that have ***ACK***ed at the start of execution. A more refined version would be throttled by various limits on simultaneous execution and peak initiation, so we may not start all of the nodes at once. There will also be late arriving nodes who **ACK** after execution begins. This implies that the execution tracker will need to be able to handle streaming nodes through the running state. This would also be helpful for pipelined job execution. Nodes reply to the **START** message with a **STARTED** acknowledgement

5.3.5 Completion

As nodes complete, we mark them as completed and **OK** state, or **FAILED** with an error state derived from the command.

The job is marked as complete when all nodes report in as **OK** or **FAILED**, or the overall job times out.

5.3.6 Cleanup and error handling.

Throughout the execution of the job, nodes may fail in various ways. The server will mark them **FAILED**. Nodes cannot be relied on to be updated as failed, or know that their role in the job is over, so a process is needed to insure that node cleanup happens properly. For example if a node ***NACK***s because it thinks it is still running a job that is over, we should be able to reset it and make it available for new work.

The pushy client does not retain job state over reboots. The client reports its current state via heartbeat messages, including the current job ID and state. The server should check the heartbeat message, and send an abort command if the node is running a dead job.

The overall philosophy for error handling should be fail fast, and don't try to do exotic recovery code. Instead of handling every possible case and trying to recover the workflow should support easily issuing jobs based on the failure status. If the commands are idempotent, they can be rerun harmlessly.

If a node crashes or restarts, that terminates execution of the job. We do not attempt to resume jobs afterwards. We should distinguish controlled restarts from outright crashes, and signal the server when we detect a shutdown in progress.

If a node disappears (loses heartbeat), it should be marked **FAILED**. If the node returns, the server should send an explicit **ABORT** message. If it has not started the reason should be 'down'. If it has started the reason will be 'crashed while executing'. While sometimes nodes might reconnect, and even have successful completed status it is simpler to treat them as failed.

5.3.7 Modifying running jobs

There are reasons to allow the currently running job to be altered in flight.



- **Aborting**
Jobs must be able to be aborted after they've started. This is accomplished by sending an abort message to each node in the job that has not already completed. If a node already started, it is marked 'aborted'. No guarantees can be made about the progress or state of nodes which have already started. If the node hasn't already started, it is marked 'not started'. In future versions we may want to have a distinction between hard and soft aborts, where a soft abort allows commands that have started to run to completion. It may also be worth allowing aborts to apply to a subset of the nodes in the job.
- **Adding nodes to a running job**
In future versions we may want to add nodes to a running job. If searches are made more dynamic, or entity groups are dynamic sets, we will want to update the job with new nodes, and have them go through the same life cycle. This would require the streaming mode of command execution.
- **Changing parameters**
In future versions we will want to allow modification the timeouts and quora requirements of a started job. For example, we may have a job running slower than expected; it's making forward progress but is at risk of timing out. The user very likely would like to alter the job time out to prevent the job from ending in a failed with timeout state.

5.3.8 Persistence and lifetime

Jobs need to be persistent on the server side; we should not 'forget' a job in progress even if the server restarts. However, clients may lose state at any time, and we must be prepared to handle it. Jobs (and the collection of node statuses in the job) persist until explicitly deleted.

5.4 Command Vocabulary

The first version of the protocol will use a restricted command vocabulary with the option of arbitrary commands. Which commands to allow is a policy decision, and probably should be configurable on a per organization basis.

- **chef-client** This command causes a managed node to initiate a chef-client run. When the one shot runlist feature is added to chef-client we will want to allow that as an option
- **ohai** This command tells a managed node to run ohai and update its node data.
- **arbitrary command?** This is the most general possible solution, and something we should consider for the long term; apparently (according to Lamont) just about every company has some provision for this in their infrastructure. Many interesting security issues appear, including what UID to run under (root? the user id that knife ssh would use?) access control problems, etc.



Perhaps this needs to have a specific ACL right in authz separate from the rest of the commands. At minimum, this should be configurable on a per-org basis, preferably via the discovery REST endpoint.

- **sleep n** This command tells a managed node to wait n seconds and then reply with success. This is intended for testing.
- **dummy_job PFAIL DURATION** This would be a more sophisticated version of the sleep command. This command tells a managed node to wait DURATION seconds and either fail with a probability of PFAIL or succeed. This would ease testing of the system failure cases.
- **abort JOBID** (TODO Does this belong here? Is this really a command?) The final product will need some mechanism to cancel/abort a job in progress. The job id may not be necessary since there will only be one job running per client. (However it might be nice if the client and server get out of sync, and the client thinks its running a different job than the server does)

5.4.1 special command types

There are a few types of commands that have interesting properties, and should be considered in future versions.

- **Informational jobs.** While most commands are run for their side effects, we may want to run some sort of information only command and return its results for future jobs. For example we could run a job that checks for a kernel version and returns the result. A new job could be started to update the nodes for which the result was a particular version. The simplest kind would simply return a boolean predicate, but more sophisticated queries could be added. However this risks turning into a generalized search language.
- **Restarting** Some commands may cause restarts on their own (e.g. running some MSI installers on Windows). The client may want to detect a proper system shutdown and report that to the job runner. Resumption of the command after shutdown is tricky, and we should probably limit expectations to commands whose last actions are restarts. Managing this on diverse platforms will require a bit of work.

Otherwise we may also want an explicit restart command in our vocabulary. This would allow chaining of jobs where there is work to do prior to the restart, a restart, and work that must be accomplished after the restart.

5.5 Client and server messages

5.5.1 Basic message format

The job command message is extremely similar to the heartbeat message. It consists of two ZeroMQ frames. The first frame contains the signature version and signature of the message.



The second frame contains the command payload for the job formatted as a JSON hash. All messages contain the following fields:

- A timestamp (used for signature verification)
- a message type field

- Client messages

All client messages add fields for:

- the node name of the sender
- the client name of the sender
- The organization name for the client.

```
{
  "timestamp": timestamp
  "type": "ready",                # message type
  "node": "node_name",
  "client": "client_name",        # OPC host name
  "org": "organization",
}
```

- Server messages

All server messages add fields for:

- The FQDN of the server
- A job_id, if relevant.

- Message formats

- Ready

Whenever a client starts up, and whenever it regains server heartbeat when idle, it sends a ready message. The message type is 'ready', and no other information is needed.

- Prepare

When a server starts a job it sends a message of type 'prepare', along with a command.

On receipt of this message client will reply with an ACK message and will be marked as ready for execution. If the client is currently executing a job, it will reply with a NACK message and the server will mark that node as ineligible to start the command.



```
{
  "server": "opc1.foo.com",          # OPC host name
  "job_id": "1234",                  # job id
  "type": "prepare",                 # message type
  "command": "chef-client"           # command to execute
}
```

- ACK/NACK messages
When a client receives a prepare message from the server, it replies with an **ACK** or **NACK** message type indicating whether it is ready for a job. The `job_id` field is set to the `job_id` of the command received. If the response is a **NACK** it also returns the reason, such as reason 'running' with 'currently_executing' set to the job id.
- START message
The start message begins execution on a node. It consists solely of the job id to start.
- FINISHED message
When a node stops or completes execution of a command, it returns the state (OK, FAILED, ABORTED) (TODO refine)
- ABORT message
At any point in the execution cycle the server can send an ABORT message. This causes immediate termination of any command execution.

5.6 Client command state machine



State	Event	Action	Next State
Startup	Detect clean shutdown		Pending
	Previously running message	Send job failed msg	Pending
	Hard shutdown detected	Send crash msg	Pending
	Defer message processing until		
Pending	Server heartbeat found		Ready
Ready	entry to state	On entry send ready message	
	Prepare message	Send ACK	Waiting (Job Id)
	Status message	Send Status	Ready
	Abort message	Ignore	Ready
	Start message	Send Start failed	Ready
	Server heartbeat loss		Pending
Waiting	Prepare message (job id matches)	Send ACK	Waiting
	Prepare message (job id doesn't)	Send NACK	Waiting
	Status message	Send Waiting info	Waiting
	Abort message	Send aborted (from waiting)	Ready
	Start message (job id matches)	Send Started, exec message	Running
	Start message (job id doesn't)	Send Error	Waiting
	Timeout on ready period	Send Error	Ready
	Server restart detected	Send Error	Ready
Running	Prepare message	Send NACK	Running
	Status message	Send Running info	Running
	Abort message	Send aborted	Terminating
	Start message (job id matches)	Ignore	Running
	Start message (job id doesn't)	Send error (can't start)	Running
	Current exec'd command succeeds	Send completed, success	Ready
	Current exec'd command fails	Send completed, message failed	Ready
	Current command exceeds time bound	Send completed, timed out	Terminating
	Server restart detected	Send Running info	Running
Terminating	Prepare message	Send NACK	Terminating
	Status message	Send Terminating info	Terminating
	Abort message	Send error (can't abort)	Terminating
	Start message	Send error (can't start)	Terminating
	Current exec'd command succeeds	Send completed, success	Ready
	Current exec'd command fails	Send completed, failed	Ready

TODO:

- Server heartbeat loss
- Server heartbeat return



- Server restart detected
- Stuck process detected (more complicated on windows)
- Add timeout for waiting state.

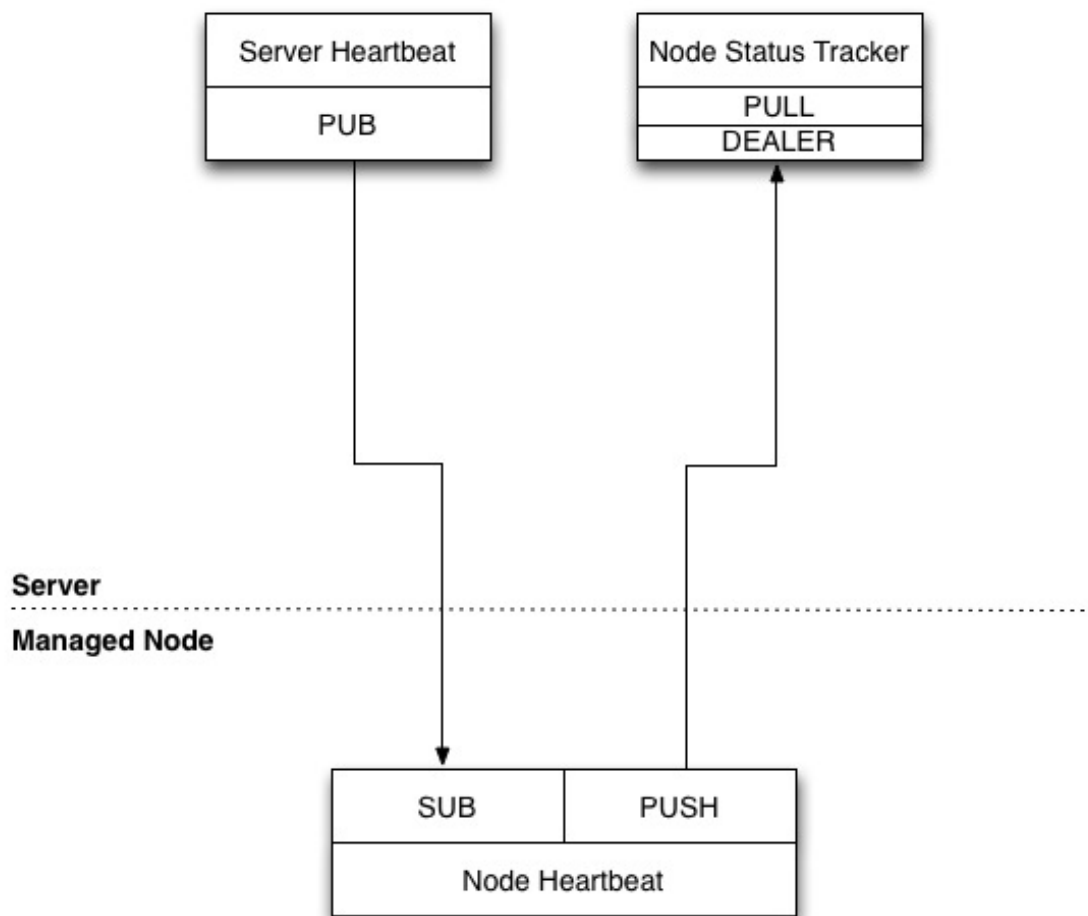


Figure 1: ZeroMQ sockets